The Stanford `libcs106` library, Academic Year 2024-25

`#include "grid.h"`

`class Grid<ValueType>`

This class stores an indexed, two-dimensional array. Rows and columns of the grid are accessed by 0-based indexes.

The following code, for example, creates an identity matrix of size **`n`**, in which the elements are 1.0 along the main diagonal and 0.0 everywhere else:

```
Grid<double> createIdentityMatrix(int n) {
   Grid<double> matrix(n, n);
   for (int i = 0; i < n; i++) {
      matrix[i][i] = 1.0;
   }
   return matrix;
}
```

Constructor

`Grid()`	O(1)	Initializes a new empty 0x0 grid.
`Grid(nRows, nCols)`	O(N)	Initializes a new grid of the given size.
`Grid(nRows, nCols, value)`	O(N)	Initializes a new grid of the given size, with every element set to the specified value.

Methods

`clear()`	O(1)	Removes all elements from this grid.
`equals(grid)`	O(N)	Returns **`true`** if the two grids contain the same elements.
`fill(value)`	O(N)	Sets every element in this grid to the given value.

get(*row*, *col*)
get(*gridLocation*)
	O(1)	Returns the element at the specified **row**/**col** or **gridLocation** in this grid.

inBounds(*row*, *col*)
inBounds(*gridLocation*)
	O(1)	Returns **true** if the specified **row**/**col** or **gridLocation** is inside the bounds of this grid.

isEmpty()	O(1)	Returns true if this grid has 0 rows and/or 0 columns.
locations()	O(1)	Returns a range containing all GridLocations for this grid.
mapAll(*fn*)	O(N)	Calls the specified function on each element of this grid.
numCols()	O(1)	Returns the number of columns in this grid.
numRows()	O(1)	Returns the number of rows in this grid.
resize(*nRows*, *nCols*)	O(N)	Reinitializes this grid to have the specified number of rows and columns.

set(*row*, *col*, *value*)
set(*gridLocation*, *value*)
	O(1)	Replaces the element at the specified **row**/**col** or **gridLocation** in this grid with a new value.

size()	O(1)	Returns the total number of elements in this grid.
toString()	O(N)	Returns a printable single-line string of this grid.
toString2D()	O(N)	Returns a printable 2-D string representation of this grid.

Operator

for (ValueType elem : grid)	O(N)	Iterates through the elements in a grid in row-major order.
grid[*row*][*col*] *grid*[*gridLocation*]	O(1)	Overloads **[]** to select elements from this grid by row/col or GridLocation.
grid1 == *grid2*	O(N)	Returns **true** if **grid1** and **grid2** contain the same elements.
grid1 != *grid2*	O(N)	Returns **true** if **grid1** and **grid2** are different.
ostream << *grid*	O(N)	Outputs the contents of the grid to the given output stream.
istream >> *grid*	O(N)	Reads the contents of the given input stream into the grid.

Constructor detail

`Grid();`
`Grid(int nRows, int nCols);`
`Grid(int nRows, int nCols, const ValueType& value);`

Initializes a new grid. The first form of the constructor creates an empty grid that contains zero rows and columns. The client must subsequently call **`resize`** to set the dimensions.

The second form of the constructor is more common and creates a grid with the specified number of rows and columns. Each element of the grid is initialized to the default value for the type.

The third form also fills every location of the grid with the given value.

The second and third constructors signal an error if a negative number of rows or columns is passed.

Usage:

```
Grid<ValueType> grid;
Grid<ValueType> grid(nRows, nCols);
Grid<ValueType> grid(nRows, nCols, value);
```

Method detail

`void clear();`

Sets every value in the grid to its element type's default value.

Usage:

```
grid.clear();
```

`bool equals(const Grid& grid) const;`

Returns **`true`** if the two grids are the same size and contain exactly the same element values. Identical in behavior to the **`==`** operator.

Usage:

```
if (grid.equals(grid2)) ...
```

`void fill(const ValueType& value) const;`

Sets every element in this grid to the given value. The entire contents of the grid are replaced with this value in every location.

Usage:

```
grid.fill(value);
```

`ValueType get(int row, int col);`
`ValueType get(GridLocation loc);`
`const ValueType& get(int row, int col) const;`
`const ValueType& get(GridLocation loc) const;`

Returns the element at the specified **`row`**/**`col`** location or **`GridLocation`** in this grid. This method signals an error if the specified location is outside the grid boundaries.

Usage:

```
ValueType value = grid.get(row, col);
ValueType value = grid.get(gridLocation);
```

`bool inBounds(int row, int col) const;`
`bool inBounds(GridLocation loc) const;`

Returns **`true`** if the specified **`row`**/**`col`** location or **`GridLocation`** is inside the bounds of the grid.

Usage:

```
if (grid.inBounds(row, col)) ...
if (grid.inBounds(gridLocation)) ...
```

`bool isEmpty() const;`

Returns **`true`** if the grid does not contain any rows or columns (size 0x0).

Usage:

```
if (grid.isEmpty()) ...
```

`GridLocationRange locations() const;`

Returns a range of all GridLocations found in this grid. This allows a nice abstraction for looping over all of the grid values using a single for loop.

Usage:

```
for (GridLocation loc: grid.locations() {
    ValueType val = grid[loc];
}
```

`void mapAll(std::function<void (const ValueType&)> fn) const;`

Calls the specified function on each element of the grid. The elements are processed in ***row-major order,*** in which all the elements of row 0 are processed, followed by the elements in row 1, and so on.

Usage:

```
grid.mapAll(fn);
```

`int numCols() const;`

Returns the number of columns in the grid.

Usage:

```
int nCols = grid.numCols();
```

`int numRows() const;`

Returns the number of rows in the grid.

Usage:

```
int nRows = grid.numRows();
```

`void resize(int nRows, int nCols, bool retain = false);`

Reinitializes the grid to have the specified number of rows and columns. Each element of the newly resized grid is initialized to the default value for the type. If the optional **`retain`** argument is true, it retains whatever previous grid contents can be. If **`retain`** is false or not given, all previous contents are discarded.

This function signals an error if a negative number of rows or columns is passed.

Usage:

```
grid.resize(nRows, nCols);
```

`void set(int row, int col, const ValueType& value);`
`void set(GridLocation loc, const ValueType& value);`

Replaces the element at the specified `row`/`col` location or `GridLocation` in this grid with a new value. This method signals an error if the specified location is outside the grid boundaries.

Usage:

```
grid.set(row, col, value);
grid.set(gridLocation, value);
```

`int size() const;`

Returns the total number of elements in the grid, which is equal to the number of rows times the number of columns.

Usage:

```
int sz = grid.size();
```

`string toString() const;`

Returns a printable string representation of this grid, such as `"{{r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2}}"` for a 2x3 grid.

Usage:

```
string str = grid.toString();
```

`string toString2D() const;`

Returns a printable 2-D string representation, such as the following for a 4x3 grid:

```
"{{r0c0, r0c1, r0c2},\n
 {r1c0, r1c1, r1c2},\n
 {r2c0, r2c1, r2c2},\n
 {r3c0, r3c1, r3c2}}"
```

Usage:

```
string str = grid.toString2D();
```

Operator detail

`ValueType operator[];`

Overloads `[]` to select elements from this grid. Can select elements by single argument of GridLocation or pair of brackets to select by row, then column. This extension enables the use of traditional array notation to get or set individual elements. This method signals an error if the `row` and `col` arguments are outside the grid boundaries.

Usage:

```
grid[row][col]
grid[gridLocation]
```

`for (ValueType elem : grid)`
`for (ValueType& elem : grid)`

The range-based for loop can be used to iterate through the elements in a collection. Iteration over a grid accesses the grid elements in row-major order.

Usage:

```
for (ValueType elem : grid) {
    cout << elem << endl;
}

for (ValueType& elem : grid) { // if reference type, elements are mutable
    elem *= 2;
}
```

`ostream& operator<<(const Grid& grid);`

Outputs the contents of **`grid`** to the given output stream. The output is in the form **`{ {r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2} }`** where elements are listed in row-major order.

Usage:

```
    cout << grid << endl;
```

`istream& operator>>(Grid& grid);`

Reads the contents of the given input stream into **`grid`**. Any previous contents of the grid are replaced. The input is expected to be in the form **`{ {r0c0, r0c1, r0c2}, {r1c0, r1c1, r1c2} }`** where elements are listed in row-major order. If unable to read a proper grid from the stream, the operation results in a stream fail state.

Usage:

```
    if (infile >> grid) ...
```
